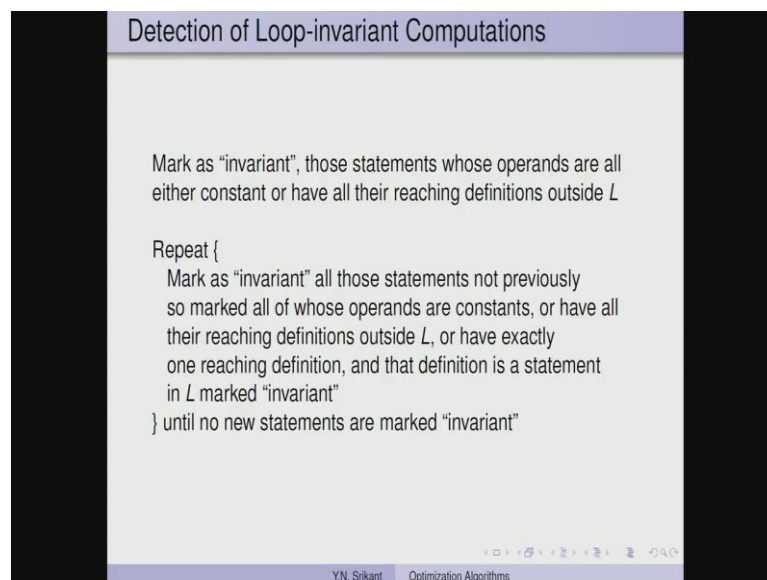


**Principles of Compiler Design**  
**Prof. Y. N. Srikant**  
**Department of Computer Science and Automation**  
**Indian Institute of Science, Bangalore**

**Lecture - 36**  
**Introduction to Machine-Independent Optimizations Part-6**

Welcome to part 6 of the lecture on Machine Independent Optimizations. Today we are going to discuss a machine independent optimizations and continue with static single assignment form and so on.

(Refer Slide Time 00:35)



**Detection of Loop-invariant Computations**

Mark as "invariant", those statements whose operands are all either constant or have all their reaching definitions outside  $L$

Repeat {  
  Mark as "invariant" all those statements not previously so marked all of whose operands are constants, or have all their reaching definitions outside  $L$ , or have exactly one reaching definition, and that definition is a statement in  $L$  marked "invariant"  
} until no new statements are marked "invariant"

Y.N. Srikant    Optimization Algorithms

In the last part of this lecture, we discussed global common sub expression in elimination and copy propagation and along with it we also saw a simple version of constant propagation. Today, we will see how to perform a major optimization called the loop invariant computation detection and code motion.

(Refer Slide Time 01:10)

The slide is titled "Loop Invariant Code motion Example". It is divided into two columns. The left column is labeled "Before LIV code motion" and the right column is labeled "After LIV code motion".

**Before LIV code motion:**

```
t1 = 202
i = 1
L1: t2 = i > 100
    if t2 goto L2
    t1 = t1 - 2
    t3 = addr(a)
    t4 = t3 - 4
    t5 = 4 * i
    t6 = t4 + t5
    *t6 = t1
    i = i + 1
    goto L1
L2:
```

**After LIV code motion:**

```
t1 = 202
i = 1
    t3 = addr(a)
    t4 = t3 - 4
L1: t2 = i > 100
    if t2 goto L2
    t1 = t1 - 2
    t5 = 4 * i
    t6 = t4 + t5
    *t6 = t1
    i = i + 1
    goto L1
L2:
```

At the bottom of the slide, there is a footer with the text "Y.N. Srikant Optimization Algorithms".

So, we informally we already know what loop invariant code is, but let us understand that with this example properly. So, the statements marked in red in this particular program are set to be loop invariant code, the reason is  $t_3$  has been assigned a constant value address  $a$  which is an offset for the you know array  $a$  inside the activation record and  $t_4$  is nothing but,  $t_3$  minus 4 again it is a constant.

So, these two statements do not get any different values even you know during any number of iterations of this particular loop, so it seems likely that these two can be shifted to a point just outside the loop. And this is exactly what is called as code motion; these are set to be loop invariant code and moving them outside is the code motion part. So, after code motion we would get a program in this form this is more efficient, obviously because these two statements execute only once and they do not execute as many times as the loop does, but moving you know detecting invariant code has a many steps.

And similarly code motion should satisfy many conditions before, the code can be moved out, so in the case of loop invariant code detection, the algorithm is quite simple. So, let me explain the algorithm with this example before we go to the text of the algorithm, to begin with initially, when the algorithm starts we mark the statements, which have some  $X$  equal to you know some constant type of computation or  $X$  equal to

some value and that value variable and that variable actually has a reaching definition only outside the loop.

In both these cases, the statement is not going to alter its values during the iterations of the loop, so therefore, such statements such as this you know either assigned a constant value or assigned some expression, whose operands have reaching definitions strictly outside the loop. In both these cases, we marked them as loop invariant in the first step, so once such statements are marked as loop invariant, for example here  $t_3$  equal to address  $a$ .

In the second iteration of the algorithm, we would mark other statements, which are dependent on such loop invariant statements, for example here  $t_4$  equal to  $t_3$  minus 4, involves  $t_3$  and the statement corresponding to  $t_3$  has already been marked as loop invariant, the other operand is a constant. It is possible that instead of a constant here we could have another operand which is also marked as loop invariant or this operand could have its reaching definitions outside the loop, so such dependent statements are marked as loop invariant in the second iteration. The third iteration considers the statements, which were marked invariant in the previous iteration and this process continues until we cannot mark anymore statements as loop invariant.

(Refer Slide Time 04:58)

Loop-Invariant Code Motion Algorithm

- 1 Find loop-invariant statements
- 2 For each statement  $s$  defining  $x$  found in step (1), check that
  - (a) it is in a block that dominates all exits of  $L$
  - (b)  $x$  is not defined elsewhere in  $L$
  - (c) all uses in  $L$  of  $x$  can only be reached by the definition of  $x$  in  $s$
- 3 Move each statement  $s$  found in step (1) and satisfying conditions of step (2) to a newly created preheader
  - provided any operands of  $s$  that are defined in loop  $L$  have previously had their definition statements moved to the preheader

Y.N. Srikant Optimization Algorithms

So, that is what is stated in the formal version of this algorithm here mark as invariant, those statements whose operands are all either constant or have all their reaching

definitions outside L, so this is the first step and then we have a loop. Now, mark as invariant all those statements not previously, so marked all of whose operands are constants or have all their reaching definitions outside L or have exactly one reaching definition and that definition is a statement in L marked as invariant, so this is the dependence part which I explain.

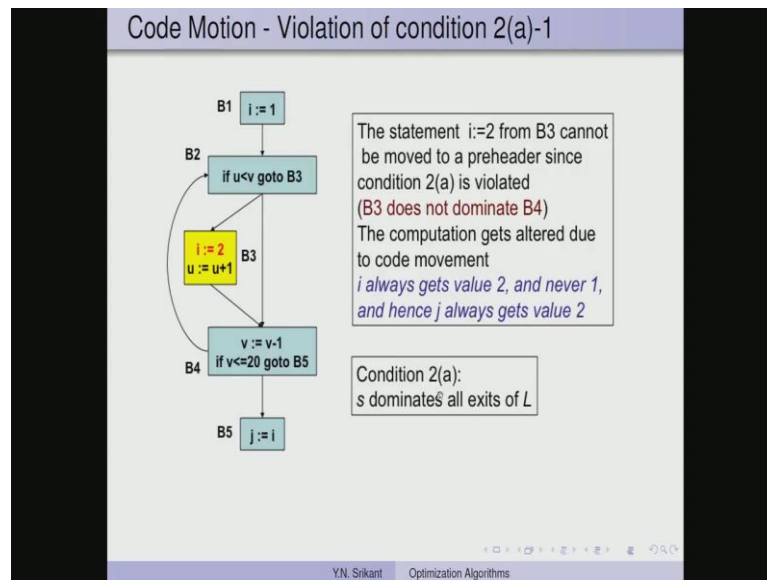
So, this loop goes on until no new statements are marked as invariant, so this is the detection of loop invariant computation, but just detecting computations as loop invariant does not take us anywhere you know, so it does not actually improve the code we should be able to move the code outside the loop. So, to do that we have a couple of you know conditions to check. The algorithm is simple first of all find loop invariant statements and then for each statement S defining X found in step 1, check these three conditions.

And then we will come to the details of these conditions very soon move each statement S found in step 1 and satisfying the conditions of step 2 to a newly created preheader just before the loop. The only requirement here is provided any operands of S that are defined in the loop L have previously had their definition statements moved to the preheader. So, in other words if we are moving a statement S, this statement would have its operands probably defined within the loop L.

So, if So we have to move those statements, which supply operands to the statement S, you know they must be move to the preheader earlier, than the statement S; in other words the we should maintain the order in which the statements execute. Now coming to the conditions that the statement S must satisfy, the first condition is S is in a block that dominates all exits of L. The second condition is X is not defined elsewhere in L and the third condition is all uses in L of X can only be reached by the definition of X in S.

So, I am going to show you know examples, where violation of any one of these conditions can lead to incorrect code motion, so it is possible to you know show that these are all requirements for code motion. And that is actually that proof is available in literature, but we are not going to discuss the proof here I am going to give you examples.

(Refer Slide Time 08:20)



So, condition to a states that  $S$  dominates all exits of  $L$ , so let us take the program which is here, so in this program the block B 3 has been highlight it and specially the statement  $i$  equal to 2 is what we are targeting. So,  $i$  equal to 2 is a statement, which can be marked as loop invariant quite easily, because it has only constant operands.

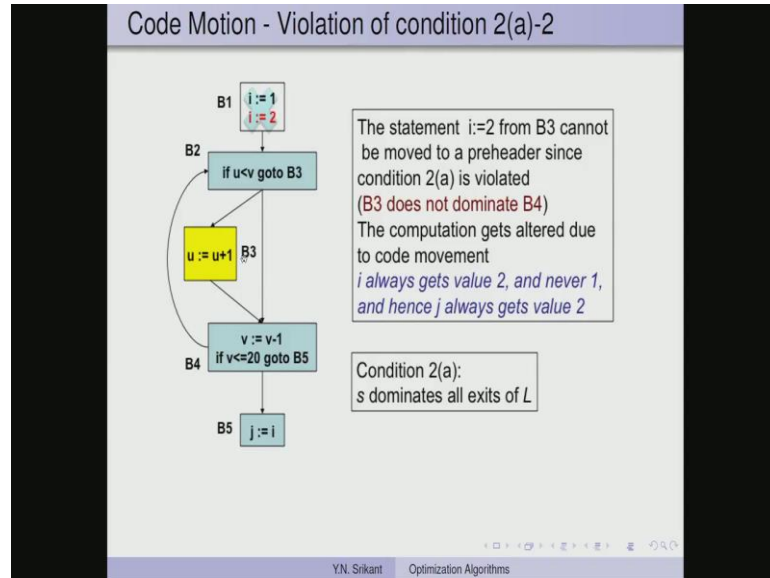
So, once having done that can we move this statement to outside the loop, so at to this one created a preheader and put it in the preheader can we do that the answer is no because condition to a which state that  $S$  dominates all exits of  $L$  is violated here. So, for example, if you consider B 3, then it is in this particular loop, which to B 3 and B4, B 4 is the for the loop and B 3 definitely, look at all the entry paths from the starting point B 1 to B 4.

So, we can actually go through the path B 1, B 2, B 4 without going through B 3, therefore B 3 does not dominate B 4, so this is a violation of condition two a. Let us see what happens if we still move the statement  $i$  equal to 2 to a preheader just outside the loop. So, the statement will be immediately after  $i$  equal to 1, so in this case, because  $i$  equal to 2 will be immediately after  $i$  equal to 1,  $i$  will get the value permanently as 2, this  $i$  equal to 1 can never be achieve in practice.

So, this goes here, so  $i$  equal to 2 will be executed immediately after  $i$  equal to 1, so  $i$  will get the value permanently as 2, so in such a case once we plus B 2. Whereas, in the case we it is possible to take this path directly and in that case  $j$  will have the value 1. So, this

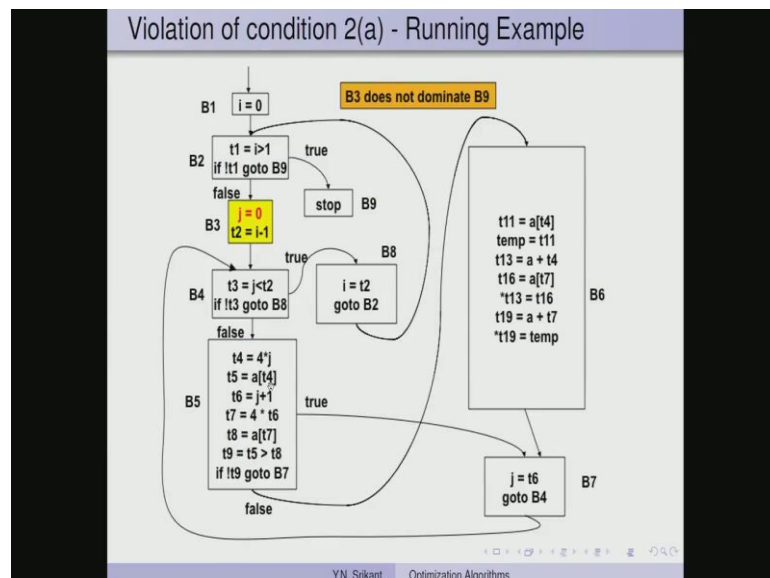
is you know clear violation of the semantics of the program, the program does not you know behave as it used to after code motion, therefore the code motion is illegal.

(Refer Slide Time 11:02)



So, now this is what I implied, here is a  $i$  equal to 1, we have moved  $i$  equal to 2, so this program is illegal.

(Refer Slide Time 11:16)

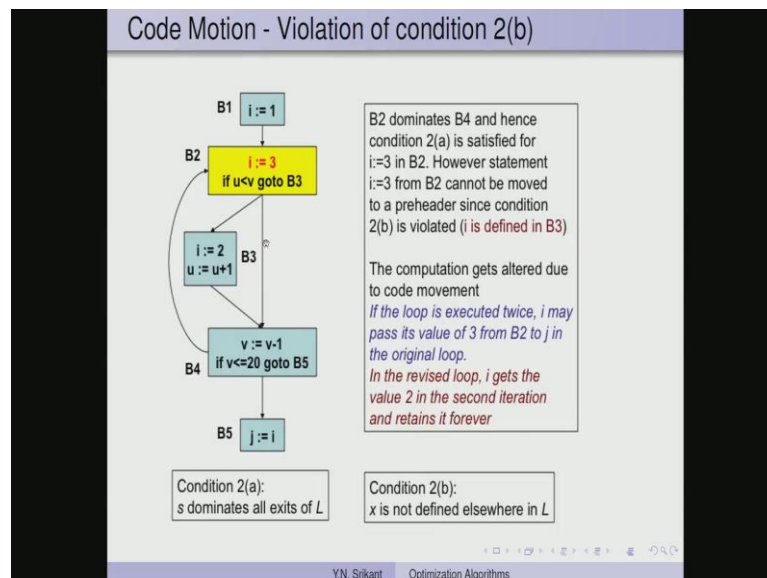


So, let us look at the running example of bubble sort, let us see if there was any possibility of code motion, so  $j$  equal to 0 is; obviously, a loop invariant computation that is very clear it is inside this loop and it has only a constant operand. Now can we move  $j$

equal to 0 to the preheader, just after B 1 outside the loop, so now the condition B 3, you know condition that B 3 which is this particular basic block, dominate the exits of L which is B 9 is violated.

So, this particular basic block does not dominate B 9 that is easy to see because from here you can go out directly to b nine without going through B 3 and as is very clear the intuitive understanding of our bubble sort program tells us that j loop is inside the i loop and j will be initialized to 0 every time. So, if we move j equal to 0 outside the loop, this initialization will not have will not happen in every iteration of i and therefore, it will be the program will not run correctly. So, that is intuitive understanding that is; obviously, correct.

(Refer Slide Time 12:53)



So, the second condition is condition to B, X is not define elsewhere in L, so consider this modified program we have B 3 in which we have i equal to 3, so i equal to 2 was here now we are not considering B 3, but we are considering B 2 which has i equal to 3.

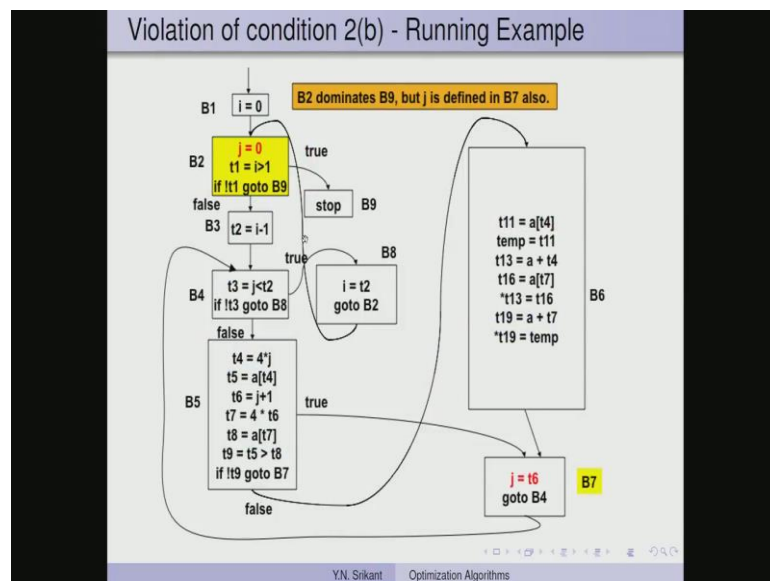
Now, is condition two a satisfied s dominates all exits of all L, so in this case B 2 definitely, you know satisfies this condition because B 2 dominates B 4, so all paths from B 1 must go through B 2 in order to reach B 4. So, that is very easy then can we now move i equal to 3 to a preheader here, so it will be after i equal to 1, again if we do that then the semantics of the program will not be correct. So, let see why so if the loop is a executed twice, that is we start here let us do this with i equal to 3 in this place itself.

So, we go through this loop once, then we actually go back and let us say then we go through this  $i$  equal to 2 and then we go back and we could actually go through this loop once more like this. So, it is possible that  $i$  get the value 2 and then goes out or after a couple of iterations, again you know we could get  $i$  equal to 3 and then go out. So, in this case  $j$  can possibly get the value 2 and sometimes it can get value 3, but suppose we  $i$  know move this computation outside the loop.

Suppose  $i$  equal to 3 is moved to this point, so  $i$  equal to three will not be executed in every iteration of the loop, now it will be executed on entry to the loop, so  $i$  equal to 1 and then  $i$  equal to 3 that is now, if we take the path this path all the time. Then that  $j$  can definitely get the value 3, but suppose this loop was executed once and then we had taken this path then  $i$  would get the value two and  $i$  can never get the value three again because  $i$  equal to 3 has been moved outside.

So, the value of  $j$  here will be always to even if we have traverse this path just once you know if we traverse it once then  $j$  becomes 2 and can never become 3, where as in this original loop even if we had traverse this path, then  $i$  would have got initialized to the on entry to the loop again. So, in the next iteration we would have initialized  $i$  to 3 and there was a chance for  $j$  to get the value 3, if this path was taken. So, the semantics of the program have changed and therefore, the transformation of moving  $i$  equal to 3 to the block just outside the loop is incorrect.

(Refer Slide Time 16:22)

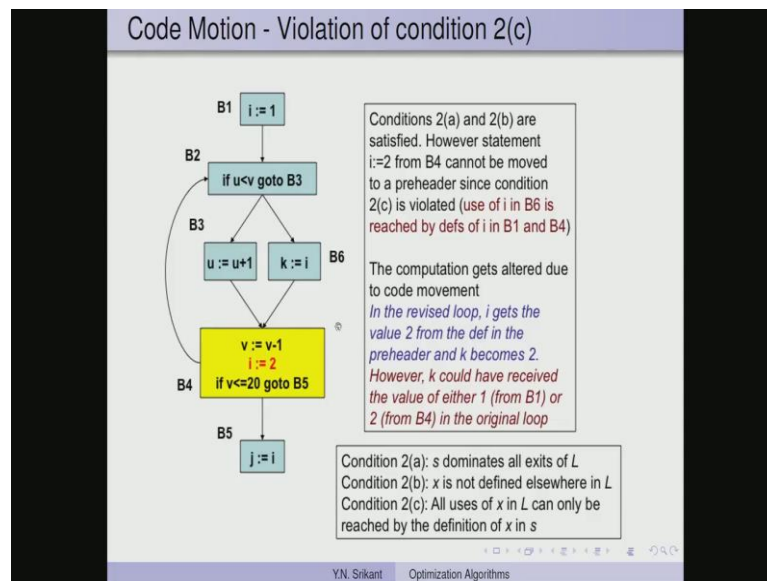




So, what happens in our running example in the running example in this case we had  $j$  equal to 0, here the semantics will not change if we move  $j$  equal to 0 here, because this is still within the loop. So, we could have actually moved  $j$  equal to 0 to this place, so like this, now this B 2 definitely dominates B 9, so condition two a is satisfied, but  $j$  is defined in B 7 also, so this is a violation of condition 2 B, so we have defined  $j$  here and we have defined  $j$  here.

In the previous example, we had defined  $i$  here and we had defined  $i$  here also, there are two definitions of  $i$  and that was a violation of condition 2 B. Here also we have a definition of  $j$  here and a definition of  $j$  here, and this you know  $j$  plus 1 which is here can be reach by this  $j$  equal to 0 and it can also be reached by this  $j$  equal to t 6. So, because of this the value of  $j$  plus 1 can be quite different, whether we get when we get  $j$  equal to 0 or the other 1. So, condition violation happens and the semantics of the program; obviously, will not be correct.

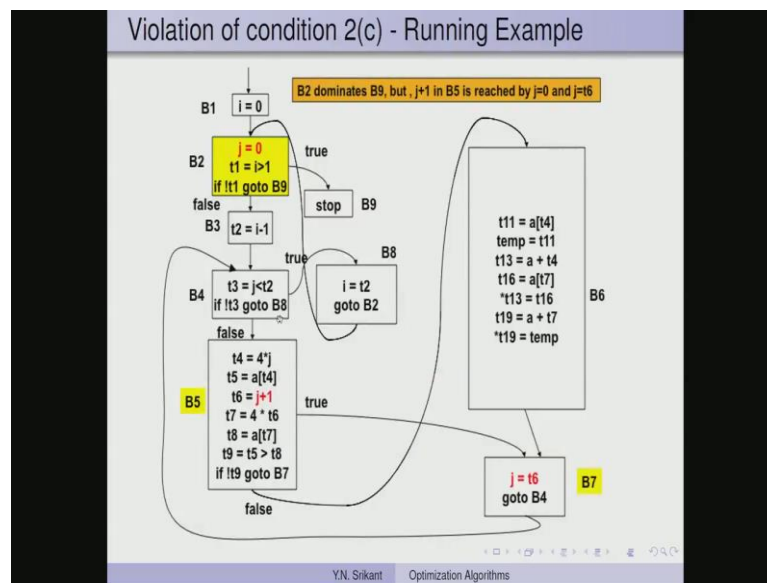
(Refer Slide Time 18:01)



Here also condition two c states that all uses of  $X$  in  $L$  can only be reached by the definition of  $X$  in  $S$ . So again let us consider this program with a slight modification  $i$  equal to 2 is a statement in the block before, now we really have  $k$  equal to  $i$  in the block B 6, this  $i$  here gets its value from this  $i$  1 possibility, it can also get this value through an iteration of the loop two second possibility.

So, there are two definitions which reach the usage of  $i$  in block B 6, so that means, the condition 2 c has been violated which says all uses in  $X$  of  $X$  in  $l$  can only be reached by the definition of  $X$  in  $S$ . So, correctly you know if you wanted to move this code to some other place only this statement should have supplied its value to  $i$  here, but we now have this and this both of them supplying value to  $i$ , so we cannot move this code  $i$  equal to two to the preheader, just after we move the code irrespective of this condition.

(Refer Slide Time 19:47)

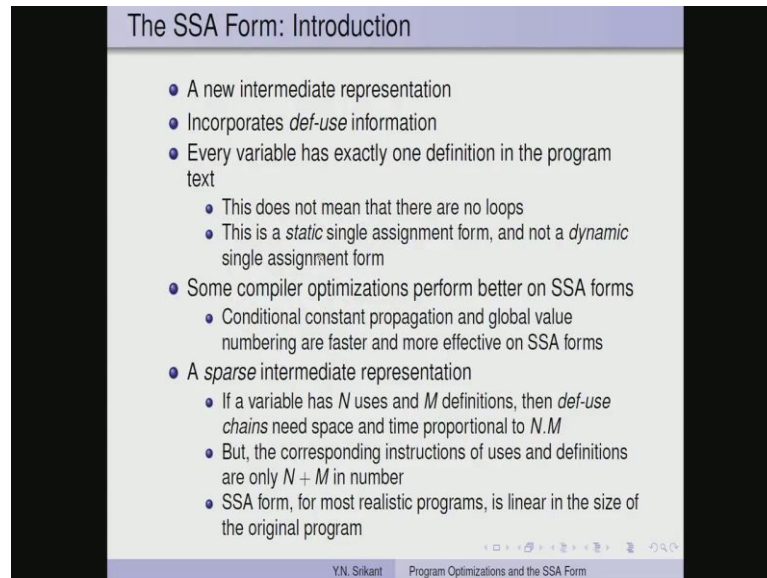


So, then again  $i$  gets the value 2 here and  $k$  will always get the value 2, it cannot get the value one at all, so this changes the semantics of the program. And therefore code movement in this case is incorrect again in the running example we have you know B 2 dominates B 9 and  $j$  plus 1, but  $j$  plus 1 in B 5 is reached by  $j$  equal to 0 and  $j$  equal to  $t$  6. So, because of this the condition 2 c is violated and we cannot move this code to outside the loop. So, that is about the loop you know machine independent optimizations on the static single assignment.

So, the lecture consists of the following parts we will discuss the static single assignment form it is definition with a few examples and we are going to discuss some of the optimizations with the SSA forms, such as dead code elimination, simple constant propagation, copy propagation and the most important of them all conditional constant propagation and constant folding. This last optimization is set to be very effective with

static single assignment form as compared to the non SSA form that is control flow graph.

(Refer Slide Time 21:12)



The SSA Form: Introduction

- A new intermediate representation
- Incorporates *def-use* information
- Every variable has exactly one definition in the program text
  - This does not mean that there are no loops
  - This is a *static* single assignment form, and not a *dynamic* single assignment form
- Some compiler optimizations perform better on SSA forms
  - Conditional constant propagation and global value numbering are faster and more effective on SSA forms
- A *sparse* intermediate representation
  - If a variable has  $N$  uses and  $M$  definitions, then *def-use chains* need space and time proportional to  $N.M$
  - But, the corresponding instructions of uses and definitions are only  $N + M$  in number
  - SSA form, for most realistic programs, is linear in the size of the original program

Y.N. Srikant Program Optimizations and the SSA Form

What is a static single assignment form this is a new intermediate representation and it incorporates defuse information, so in other words the optimization is actually incorporated into this new static single. So, every variable has exactly one definition in the program text, so that is why this is called as a static single assignment form, but I must has not to add that this does not mean that there are no loops, the program definitely can have loops, without loops you cannot write meaningful programs anywhere, we static assignment you know should be highlighted.

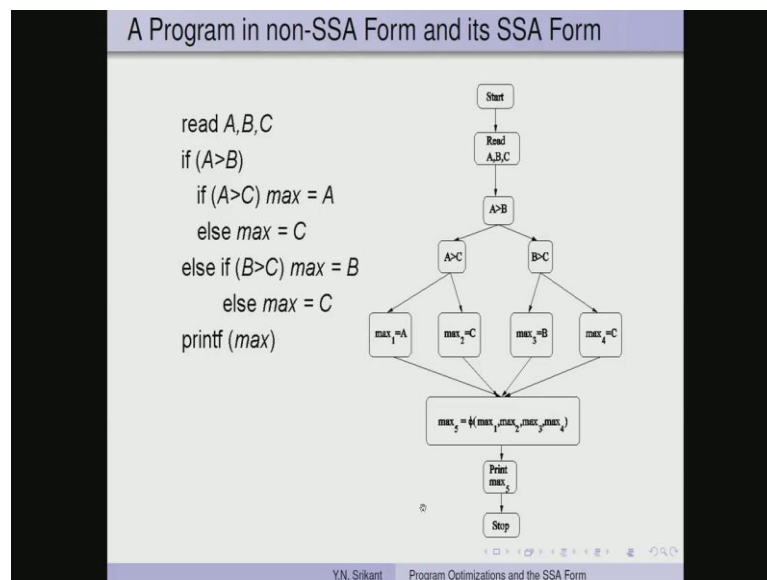
So, this is a static single assignment form and not a dynamic single assignment form in other words we are making sure that in the program text as it is given, exactly one assignment, two each variable exist it does not dynamic means in the loop the statement would be you know assigned the variable will be assigned values again and again. So, this dynamic assignment is definitely permitted multiple assignment are permitted.

But what we mean is in the program text that is static text a is to be asserted, so loops are permitted, but within the loop again the text must not have more than one assignment to the variable optimizations perform much better on the static single assignment forms. The most important of them all is the conditional constant propagation and the other one

is the global value numbering, these two are faster and more effective on the static single assignment forms.

SSA is a sparse intermediate representation, so what do you mean by sparse, if a variable  $n$  variable has  $n$  uses and  $m$  definitions, then the defuse chains requires space and time proportional to  $n \cdot m$  whereas, the corresponding instructions of uses and definitions are only  $n$  plus  $m$  in number. So, there are  $n$  uses and  $m$  definitions, so if you just add up it is  $n$  plus  $m$ , SSA form for most realistic programs is linear in the size of the original program. So, this is called as a sparse intermediate representation, so in the size of the original program.

(Refer Slide Time 24:15)



Let me give an example, this is a fairly simple example, here is the program text, so what we have done here is show you the program in non SSA form and the flow graph in SSA form. So, this is a simple max computation read A,B,C if A greater than B, then if A greater than C, max is a obvious otherwise max is C, now if B greater than C, then max equal to B, otherwise max equal to C. So, print maximum.

So, the flow graph faithfully says read A,B,C then it checks a greater than B, if true it checks a greater than C, if false it checks B greater than C, so far there is no change in the flow graph corresponding and the corresponding program in non SSA form. The change now happens now, so we have the same variable max in all the four cases static

single assignment form, we are not permitted to use the same variable again and again for assignment.

So, we would have max 1 equal to A here, you know max 2 equal to C here, max 3 equal to B here and max 4 equal to C here, so there are four assignments and each of them uses a different max variable, if we do that then we run into a problem at this point we are supposed to print the value of max and there are 4 values of max. So, which one of these max values should be printed, so this is a common problem at a join we are supposed to have a single value of a max variable to get that the static single assignment form introduces a merge or join operator called as phi.

So, this merge operator takes as many operands or parameter as the number of incoming arcs, so we have max 1 and max 4, so we have 4 of 4 parameters here, so there are 4 different values which are possible along the 4 different incoming arcs. So, it has 4 parameters and it is returned as max phi equal to because again we cannot reuse any of these names, so we create a new name and make it max phi and what is printed here is max phi.

The semantics of the merge operators states that the value of operand or the parameter which is relevant or picked up is the one corresponding to the flow of control. So if the flow of control actually happens via this path, we actually enter this basic block via this basic block, then we would have taken this edge to enter it, then this is the first edge, this is the second edge, this is the third edge and this is the fourth edge. So, if the entry happens via edge 1, then parameter 1 will be taken as the value if it is through edge 2, then the second one, if it is edge 3, the third one and if it is edge 4, then the fourth one, so that would be the value of phi.

So, this is the semantics of phi statement; obviously, this phi does not have any physical machine instruction corresponding to it you know in any machine, so usually we are required to translate it to you know copy instructions and so on and so forth. So, but that would be done at the time of machine code generation, so we are not going to worry about it during the optimization phase.

(Refer Slide Time 28:41)

The SSA Form: Introduction

- A new intermediate representation
- Incorporates *def-use* information
- Every variable has exactly one definition in the program text
  - This does not mean that there are no loops
  - This is a *static* single assignment form, and not a *dynamic* single assignment form
- Some compiler optimizations perform better on SSA forms
  - Conditional constant propagation and global value numbering are faster and more effective on SSA forms
- A *sparse* intermediate representation
  - If a variable has  $N$  uses and  $M$  definitions, then *def-use chains* need space and time proportional to  $N.M$
  - But, the corresponding instructions of uses and definitions are only  $N + M$  in number
  - SSA form, for most realistic programs, is linear in the size of the original program

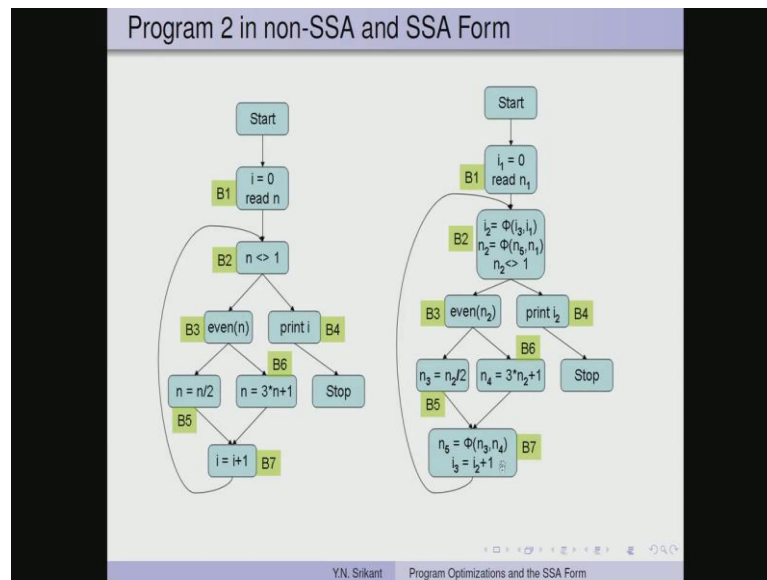
Y.N. Srikant Program Optimizations and the SSA Form

So, let us define the SSA form, now that we have seen an example a program is in a SSA form, if each use of a variable is reached exactly by one definition, so this is the static definition part. Flow of control remains the same as in the non SSA part a special merge operator phi is used for selection of values in join nodes, i already explain this not every join node requires a phi operator for every variable well the no need for a phi operator.

If the same definition of the variable reaches the join node along all incoming edges even in the previous example, we do not have you know this is a join node, we do not have any merge operator for A, B and C. Because it is not required the value of A, B and C do not change along any path, therefore there is no need for A, B and C to have merge operators at this point.

So, this is what it says no need for a phi operator, if the same definition of the variable reaches the join node along all incoming edges often an SSA form augmented with u-d and g-u, d-u change to facilitate design of faster algorithms. We will see this later translation from SSA to machine code introduces copy operations which may introduce some inefficiency, but hopefully good register allocation takes care of this inefficiency.

(Refer Slide Time 30:29)



So, now let us discuss another example of the SSA form, so this is the program in non SSA form or the flow graph in non SSA form and this is the program in SSA form. So, we have only two variables in this program  $i$  and  $n$ , so  $i$  is assigned a value and  $n$  is actually read from inputs, so that is as good as an assignment.

And then we compare  $n$  with 1, if it is true then we check whether  $n$  is even, then again if it is true then we have  $n$  equal to  $n$  by 2 if it is false we have  $n$  equal to  $3 \cdot n + 1$  both of them join here we increment  $i$  and go back. So, if this was false then we would have if  $n$  not equal to 1, we would have this was false then we would have come here print  $i$  and you know gone ahead stop, so let us see how this can be you know how the SSA form reads.

So, we have  $i$  equal to 0 here and then we also have a  $i$  equal to  $i$  plus 1, we have two assignment to  $i$  therefore, we require more than one  $i$  variable, so we have  $i_1$  equal to 0 at this point, then we have read  $n$ , so we have read  $n$ . So, that is the first variable for  $n$  there are many assignments to  $n$  again, now this is a join point at this point, we have a join, so there is a value of  $i$  which flows along this arc and there is a value of  $i$  which flows along this arc.

So, obviously this two values can be quite different and therefore, we have a 5 variable here, we have a phi operator, here the value of  $i$  let us say is  $i_3$  at this point that is from which flow along this direction and the value of  $i$  which comes along this arc is  $i_1$ . So,

we have a selection between  $i_3$  and  $i_1$ ; obviously, the variable  $i$  for  $i$  which is going to collect the value from the phi operator is also an assignment and that requires a different instance of  $i$ , so here is where we have  $i_2$ .

So, we have  $i_1$  and then  $i_2$  which just collects the value from the phi operator and then we have  $i_3$  which is the assignment in this block B 7, so similarly that takes care of  $i_2$  rather  $i$ . So, remember even though there are only two assignments to  $i$ , it does not mean we require only two different instances of  $i$ , if we had just made this  $i_1$ . And this as  $i_2$  then you know getting the appropriate value of  $i$  here through the phi statement would have made this  $i_3$  instead of  $i_2$  that is all.

So, that is nothing but just a you know naming convention and nothing more than that the naming or renaming algorithm takes care of all such minor problems, then you know there is a value of  $n$  which comes in here and a value of  $n$  which comes in because of this as well. So, assuming that a we have a value you know of  $n$  which is coming in this direction as  $n_3$  and the value which comes along this direction as  $n_4$ , we require a phi operator here to supply the value to this particular node.

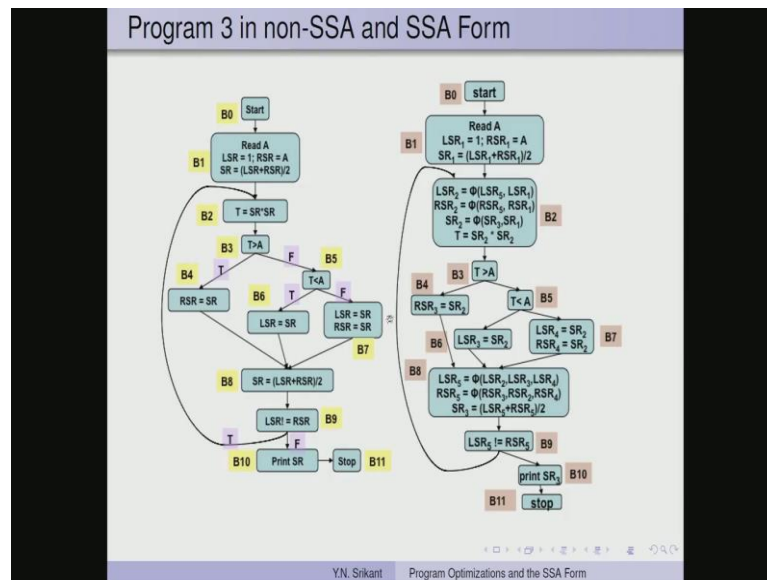
So, that would be called as  $n_5$  equal to phi of  $n_3$  comma  $n_4$ , so depending on the entry either this or this one of these will be picked up that will be called as  $n_5$ , and supply to this particular join node, so if the entry is via this arc then we pick up  $n_5$ . And if the entry via this arc we pick up  $n_1$ , so that is given the variable name  $n_2$ , so  $n_2$  is compared here instead of  $n$  a not equal to 1,  $n_2$  is compared here, then we check whether  $n_2$  is  $e_1$  and then because there is an assignment to  $n$ .

We introduce a new variable  $n_3$ , so that gets the value  $n_3$  equal to  $n_2$  by 2 and here we have to introduce a fourth variable  $n_4$  which is  $3 \star$  into plus 1 and because of these two different variables joining at this point. We require a phi operator here which picks up one of this values and supplies it to this node, so even though we had a 1 then 2 and three assignments to  $n$ .

We have introduced really you know this is  $n_1$  here is  $n_2$ , then we have  $n_3$ ,  $n_4$  and  $n_5$ , so 5 instances of the variable  $n$  have been introduced even though the original had only three assignments to  $n$ . So, the rules for introducing the phi operators and new assignments is a bit complicated we are not going to discuss the a construction of the SSA form which takes care of such difficulties.



(Refer Slide Time 36:36)



Let us look at one more example, so this is actually a straight forward program in which we have a LSR, RSR and SR as the main variables then there is A T equal to SR star SR, so then we compare T greater than A etcetera.

If it is true we assign RSR equal to SR, if T greater than A is false, then we compare T less than A, if that is true we have LSR equal to SR, otherwise we have LSR equal to SR and RSR equal to SR. There is a joint point here we compute SR as a LSR plus RSR by 2, then LSR not equal to RSR, we go back and iterate, so do not bother about the mining of the program it really does not do anything meaningful, it has been constructed just to show the SSA version.

So, here is a you LSR computation, so we have LSR 1, RSR 1 and SR 1, then we have a you know a join block here, so there is a value of LSR, RSR and SR given here and through this arc we get a version LSR, RSR and SR. Rather SR is not supplied here SR is computed now it is computed it is actually supplied. So we have SR version also coming in here, so for that we have new variables for a LSR, RSR and SR which are actually the collecting variables for the phi operators then we compute T.

So, then we go on to check T against T, we have new instances of RSR, LSR and SR generated here and once we do that you know we again require a phi function here this requires a phi function. Because there is a version of LSR coming here, which is the old

one and then a new value of a LSR comes here and another new value of LSR comes here.

So, there are three different values of LSR coming in through these 3 paths, so we require a phi operator with 3 operands, the same is true for RSR and of course, SR is computed here. So, that is a new variable as well, so this is the way the SSA form appears, it has a phi operators whenever we need to pick a corresponding value you know with each entry point.

(Refer Slide Time 39:31)

The slide is titled "Optimization Algorithms with SSA Forms" and contains a bulleted list of optimization techniques. The list includes: Dead-code elimination (with sub-points: Very simple, since there is exactly one definition reaching each use; Examine the du-chain of each variable to see if its use list is empty; Remove such variables and their definition statements; If a statement such as  $x = y + z$  (or  $x = \phi(y_1, y_2)$ ) is deleted, care must be taken to remove the deleted statement from the du-chains of  $y$  and  $z$  (or  $y_1$  and  $y_2$ )), Simple constant propagation, Copy propagation, Conditional constant propagation and constant folding, and Global value numbering. The slide footer shows the name "Y.N. Srikant" and the title "Program Optimizations and the SSA Form".

- Dead-code elimination
  - Very simple, since there is exactly one definition reaching each use
  - Examine the *du-chain* of each variable to see if its use list is empty
  - Remove such variables and their definition statements
  - If a statement such as  $x = y + z$  (or  $x = \phi(y_1, y_2)$ ) is deleted, care must be taken to remove the deleted statement from the *du-chains* of  $y$  and  $z$  (or  $y_1$  and  $y_2$ )
- Simple constant propagation
- Copy propagation
- Conditional constant propagation and constant folding
- Global value numbering

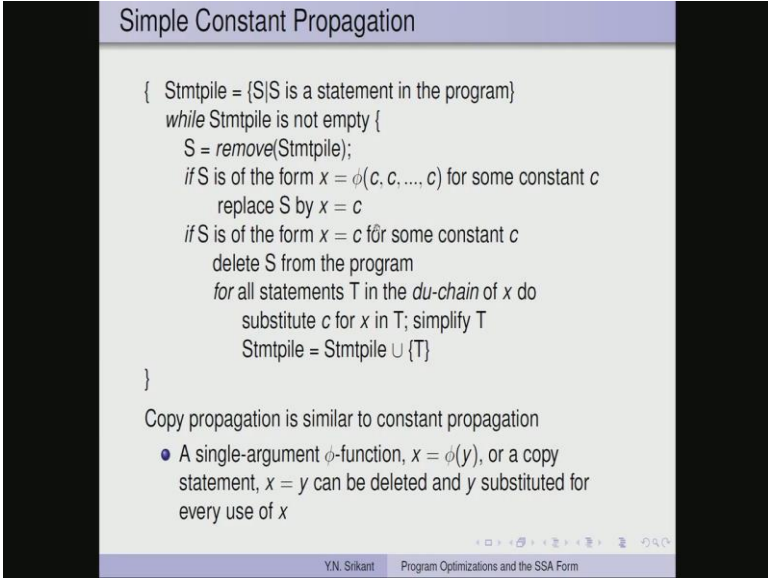
So, now let us discuss the optimization algorithms, the first one is a dead code elimination which is very simple, the reason why it becomes simpler then before is that there is exactly one definition reaching each use. By the way let me show you that here, so in this new version SSA version, this definition  $n_3$  equal to  $n_2$  by 2, reaches this use, so that is the only thing and then we have this  $n_3$ , so again that is corresponding to this definition, so we do not have more than one definition reaching any use.

So this is reached by this and then the  $n_1$  is reached by this and  $n_2$  is a new definition which will reach all these uses this and this, so exactly one definition reaches every use. So, because there is only one definition reaching each use we examine the d-u chain of each variable to see if it is use list is empty if there is no use list then; obviously, the code is dead remove such variables and their definition statements. And if a statement such as  $X$  equal to  $Y$  plus  $Z$  is deleted we must take care to remove the deleted statements from

the d-u chains of Y and Z as well, so obviously, if this goes away this Y and this Z will not be you know valid uses anymore.

So, we must remove the quadruple numbers of these statements from the definition of Y and Z, the same is true for the phi statement as well. So, let us consider the simple copy constant propagation copy propagation and then conditional constant propagation global value numbering is bit more complicated, so we are going to skip that.

(Refer Slide Time 41:42)



Simple Constant Propagation

```
{ Stmtpile = {S|S is a statement in the program}
while Stmtpile is not empty {
  S = remove(Stmtpile);
  if S is of the form  $x = \phi(c, c, \dots, c)$  for some constant  $c$ 
    replace S by  $x = c$ 
  if S is of the form  $x = c$  for some constant  $c$ 
    delete S from the program
    for all statements T in the du-chain of  $x$  do
      substitute  $c$  for  $x$  in T; simplify T
    Stmtpile = Stmtpile  $\cup$  {T}
}
```

Copy propagation is similar to constant propagation

- A single-argument  $\phi$ -function,  $x = \phi(y)$ , or a copy statement,  $x = y$  can be deleted and  $y$  substituted for every use of  $x$

Y.N. Srikant Program Optimizations and the SSA Form

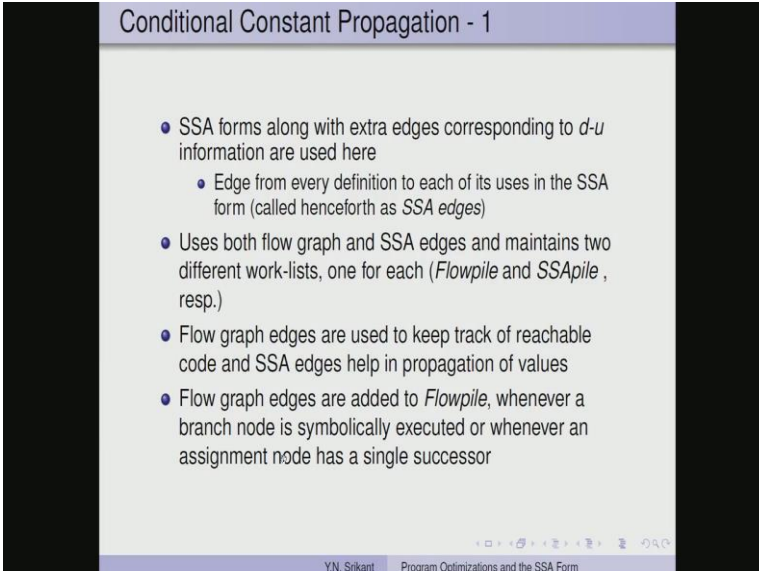
Simple constant propagation is really simple; it is actually a simplified form of the constant propagation we studied with control flow graphs. So, we form the statement pile as before S is a statement in the program. And while the statement pile is not empty we execute the loop, we remove a statement from the pile and if S is the form X equal to phi of c comma, c comma, c comma, c for some constant c, then we can replace this statement by X equal to c that is very obvious and a now if the statement is of the form X equal to c.

Either obtained by this simplification or otherwise for some constant c, then we delete S from the program because we do not have to perform any dead code elimination, we will see why for all statements T in the d-u chain of X, we substitute c for X in T. So; obviously, the check that we did in the control flow graph whether the you know whether there is more than one definition reaching X that need not be this use of X need not be done here exactly one definition reaches.

So, we can simply substitute  $c$  for  $X$  in  $T$  and we simplify  $T$ , so we can definitely do this for all the statement in  $d-u$  chain and once we do that, this  $X$  equal to  $c$  becomes redundant. Therefore, we can do that elimination here, we could not have done that in the control flow graph, now add  $T$  to this statement pile and we go back, so this is a very simple constant propagation, copy propagation is similar to constant propagation.

A single argument phi function  $X$  equal to phi  $Y$  or a copy statement,  $X$  equal to  $Y$  can be deleted and  $Y$  substituted for every use of  $X$ . We do not have to check anything here, because exactly one definition reaches the use of  $X$ , that is this definition reaches the use of  $X$  here, and no other definition can reach that point, so copy propagation is very simple here.

(Refer Slide Time 44:01)



Conditional Constant Propagation - 1

- SSA forms along with extra edges corresponding to  $d-u$  information are used here
  - Edge from every definition to each of its uses in the SSA form (called henceforth as SSA edges)
- Uses both flow graph and SSA edges and maintains two different work-lists, one for each (*Flowpile* and *SSApile*, resp.)
- Flow graph edges are used to keep track of reachable code and SSA edges help in propagation of values
- Flow graph edges are added to *Flowpile*, whenever a branch node is symbolically executed or whenever an assignment node has a single successor

Y.N. Srikant Program Optimizations and the SSA Form

Let us now, consider conditional constant propagation static single assignment form with extra edges corresponding to the definition use information are used here, so this is the augmented version of the SSA form edge, from every definition to each of its uses in the SSA form. Hence, forth we call this edges are as SSA edges, this are added to the SSA form, it uses both the flow graph and the SSA edges flow graph edge.

And the SSA edge and it maintains two different quest or work list, so one of them it does not mix up the two kinds of edges, one of them is called the flow pile which store the flow graph edges and the other one is called the SSA pile which store the SSA edges. So, flow graph edges are used to keep track of reachable code, so there flow graph edges

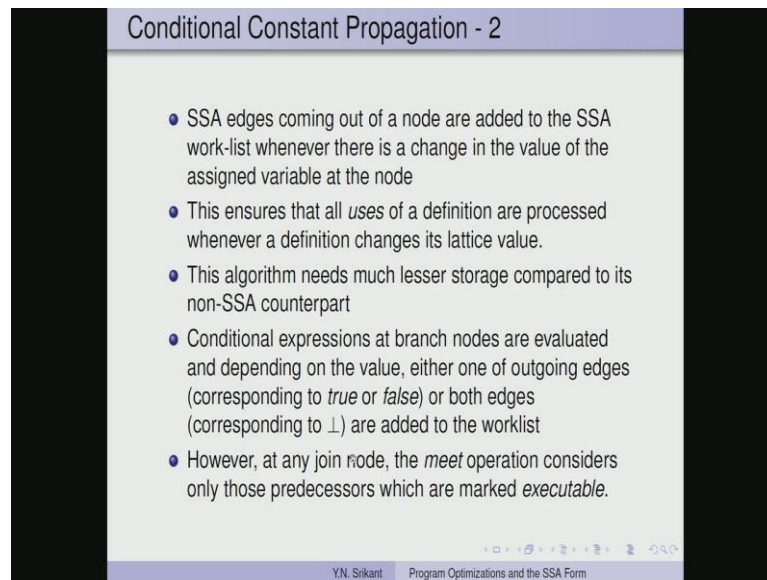
indicate you know the flow of control, so we are going to use that to keep track of reachable code.

So, if we cannot reach some code using a flow graph edge as you see in the example, then you know this is unreachable code and it can be eliminated. Then what is the purpose of SSA edges? SSA edges are useful in the propagation of values. So, what happens if it is cumbersome to traverse the control flow graph every time just to check the changes which happen in basic blocks because of a definition change, such as you know changes can be easily propagated using the SSA edge.

So, we do not have to traverse the whole control flow graph to come to a particular basic block, we can directly go to that basic block and look at the changes that happened there, so the algorithm actually becomes much faster, because of the SSA edges if we had used only flow graph edges. We could still have performed constant propagation, but now with SSA edges the program works much faster. The constant propagation program becomes much faster.

Flow graph edges are added to the flow pile, whenever a branch node is symbolically executed, so or whenever an assignment node has a single successor, so when we execute a flow graph you know a branch node we may take the true edge or the false edge in case the condition. Actually, evaluates to true or false if the value is not known then we have to actually add both the edges to the flow pile, whenever an assignment node is executed. And it has a; obviously, single successor, then that edge is added to the flow graph pile flow pile, because the successor node of the assignment node is the one which would be executed after the assignment.

(Refer Slide Time 47:35)



Conditional Constant Propagation - 2

- SSA edges coming out of a node are added to the SSA work-list whenever there is a change in the value of the assigned variable at the node
- This ensures that all *uses* of a definition are processed whenever a definition changes its lattice value.
- This algorithm needs much lesser storage compared to its non-SSA counterpart
- Conditional expressions at branch nodes are evaluated and depending on the value, either one of outgoing edges (corresponding to *true* or *false*) or both edges (corresponding to  $\perp$ ) are added to the worklist
- However, at any join node, the *meet* operation considers only those predecessors which are marked *executable*.

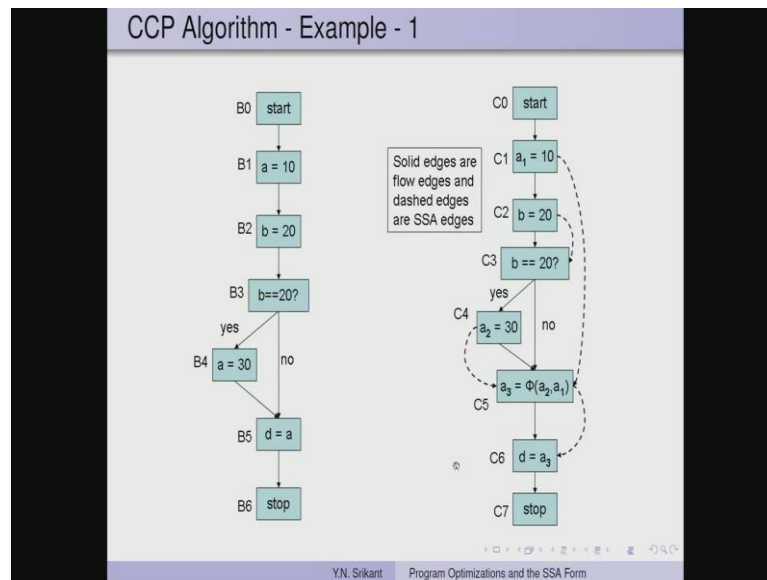
Y.N. Srikant Program Optimizations and the SSA Form

SSA nodes coming out of a node edges, rather coming out of a node are added to the SSA pile, whenever there is a change in the value of the assigned variable at the node, so this becomes a very clear as we go along. So, if there is a definition and the variable associated with that definition changes its value then the SSA edge, corresponding to that node will be added to the SSA pile, this ensures that all uses of a definition are possessed.

Whenever, the definition changes its lattice value, so the lattice value that I am referring to here is the abstract lattice semi lattice value that we had discuss before that is the top or the constant value or the bottom, so the same lattice is used by the conditional constant propagation as well. This algorithm requires much lesser storage compared to the non SSA counterpart and conditional expressions at branch nodes are evaluated and depending on the value.

Either, one of the outgoing edges corresponding true and false or both edges are added to the flow pile work list; however, at a any join node the meet operation considers only, those predecessors which are marked as executable. So, this is very important if a, if a particular node cannot be reached via a predecessors which is a you know, which is indicated by the corresponding incoming edge being not marked as a executable, then we do not have to bother about that a incoming node at all.

(Refer Slide Time 49:25)

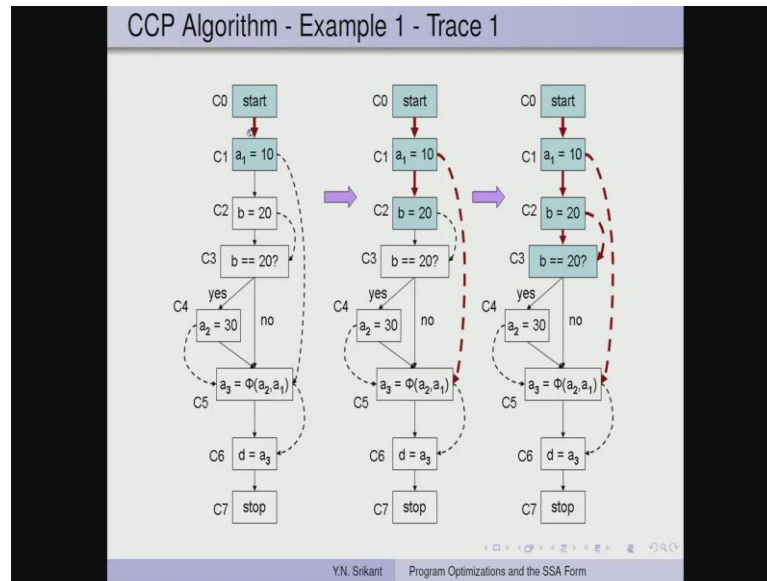


So, here is the first example, so we have start then a equal to 10 B equal to 20, we compare B with 20, if it is true then we assign a to 30, if it is false we go straight, so either way, we come here we assigned d equal to a. So, d can take the value either 30 or 10 then we stop, but if you look at it more carefully because B is a constant the check B equal to 20 actually happens to be true, so this no part will never be executed.

So, we actually go here a is always 30, so we can assign the value of d as 30 and then stop the SSA form is this, so we have a 1 equal to 10 B need not be rename, because there is one version of B, then B equal to 20 checked. So, we have another version of a here, so a equal to 30 and since we have two incoming arcs we have a 5 with a 2 and a 1 and that is assign a 3, there is a change in the value of B, so it is propagating to the next use of B.

So, that is only this node here we have a 1 equal to 10, so that value is used here, so it is actually there is a next to this node here, we have d equal to a 3, so this value is defined here. So, the edge actually links to this node etcetera, similarly this a 2 is 30, so it is linked to this node, because a 2 is used here, so this are the definition use edges are the SSA edges.

(Refer Slide Time 51:18)

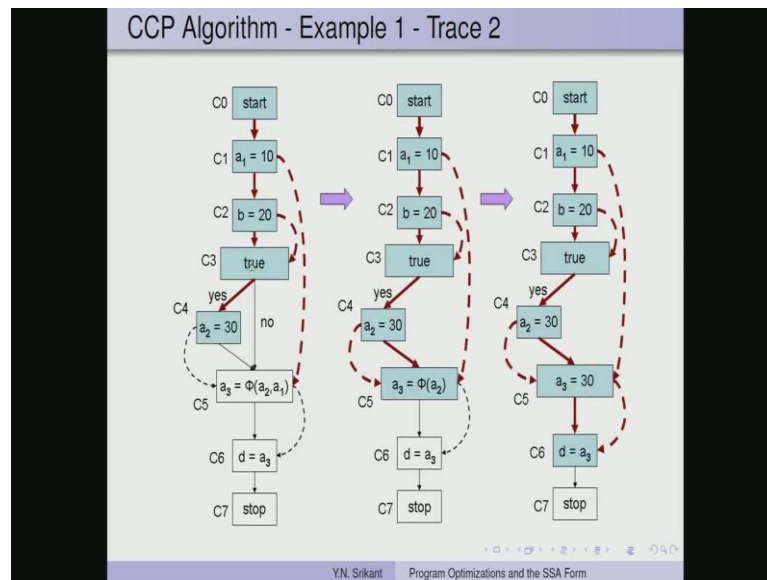


So, this let us look at the trace of this example, so we start here execute, so this edge goes on to the flow pile and then when we look at that edge we take this node and interpret a 1. So, that becomes a 1 equal to 10, so the constant value is retained, then you know it has a single successor, so this edge will be added to the flow pile. So we traverse this edge come to this node, so B equal to 20, will be added to the flow pile in the mean while this SSA edge will added to the SSA pile.

So, we come to this stage, where, we pick up this node execute it, so B is treated as a constant its value is store, so this edge is now added to the flow pile, so that brings us to this node, now because of B equal to 20, this will added to the SSA pile. So, B equal to 20 is compared, so since the value of B is known to be a constant, we can evaluate this constant as this expiration as true.



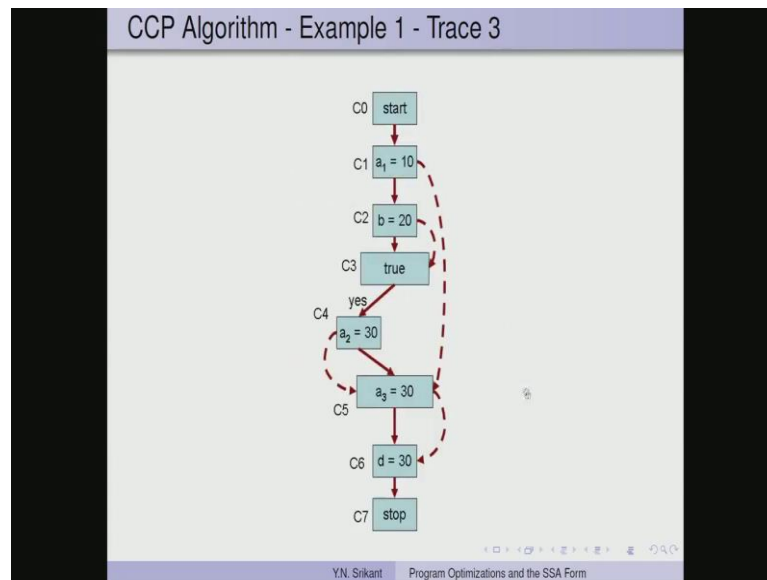
(Refer Slide Time 52:36)



So, this becomes true and therefore, we need to add only the true edge to the flow pile and the no edge false edge need not be added to the flow pile, so in this case we have this definition a 2 equal to 30 which is stored. So, a 2 is now taken as a constant, so then that activates this edge and we go to this node in this node, we have a 3 equal to phi of a 2, because we consider only those incoming edges which are marked as a executable.

So, only these edges marked as a executable, this edge is not marked as a executable, so we consider only this particular predecessor in phi with there is only one predecessor, so value of a 2 is what is given to a 3, so it can be actually evaluated as a statement a 3 equal to 30 directly. So, now we mark this edge as executable, because this is the only outgoing edge from this node, so we go to this particular node this has the value rather the statement d equal a 3, now a 3 can be determined as constant from the previous interpretation, so that would be assigned to d.

(Refer Slide Time 54:04)

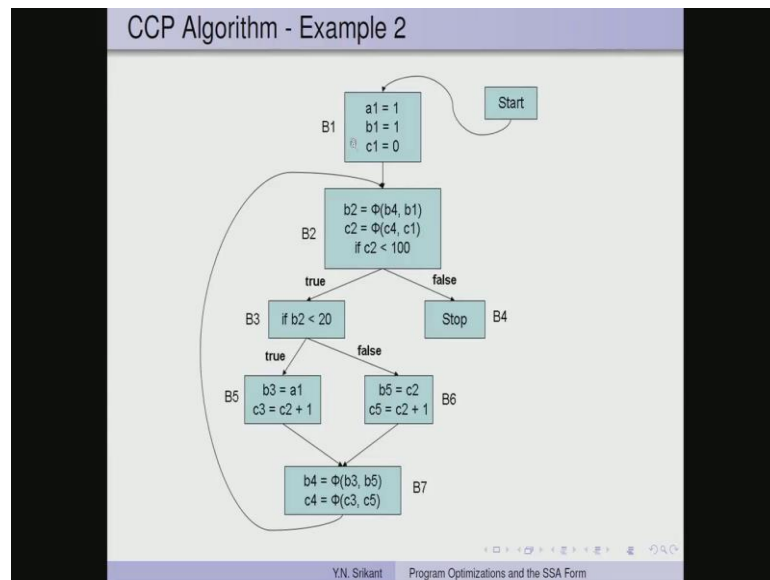


So, after all this we have a this structure, where a 1 equal to 10, B equal to 20, this is a true this is a 2 equal to 30, a 3 equal to 30. And then d equal to 30, in this particular example the SSA edges have no use, because the we have come to the end of the graph. So, flow graph edges rather the SSA edges become useful only later, so now, we can now perform a few more optimizations on this for example, we have a 1 equal to 10 here.

And then we have a 2 equal to 30 here, both of which have no influence at all on d, so this is really dead code. So, a 3 equal to 30 is the only one which is relevant to us, so d equal to 30 again, so if a 3 is not used later on we can actually delete a 1, then a 2 and a 3 all this we can we need to retain only d.

Similarly, if B is not used then this node can also be deleted, the two node has no value now, so we really have only d equal to 30, from now on if none of them other variables are used, otherwise we would have B equal to 20 as one statement. And then a 3 equal to 30 as another statement and finally, d equal to 30 as the third statement, so we would have these values these are the only nodes which are relevant in this particular example this example.

(Refer Slide Time 55:51)



I am going to explain the example and then we will discuss the details in the next part of the lecture, so here we have a loop as well. So this is the loops structure and we have many merge functions one for B 2 here another for c 2 here and then we have 1 for B 4 here another for c 4. So, in this example there are going to be a SSA edges, which are going to be useful to make the flow of value to the various basic blocks quite visible and make the processing much faster, so we will consider this in detail in the next part of the lecture, so we will stop here.

Thank you.